HANDS ON GWT

LOGIKAS
CONECTANDO IDEAS

Unidad II

“Interactuando con el Servidor”

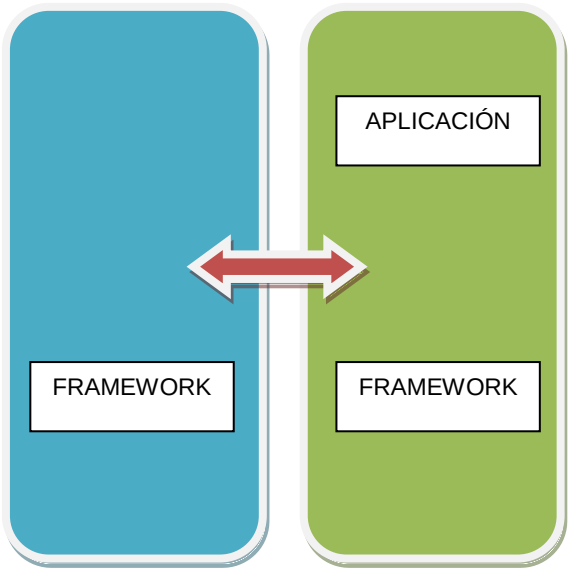
1) Introducción:

En la unidad anterior estudiamos cómo crear un proyecto en GWT y cómo funciona una aplicación GWT en el navegador. En esta unidad comprenderemos cómo interactúan las aplicaciones GWT con el servidor, explorando las diferentes alternativas de comunicación existentes.

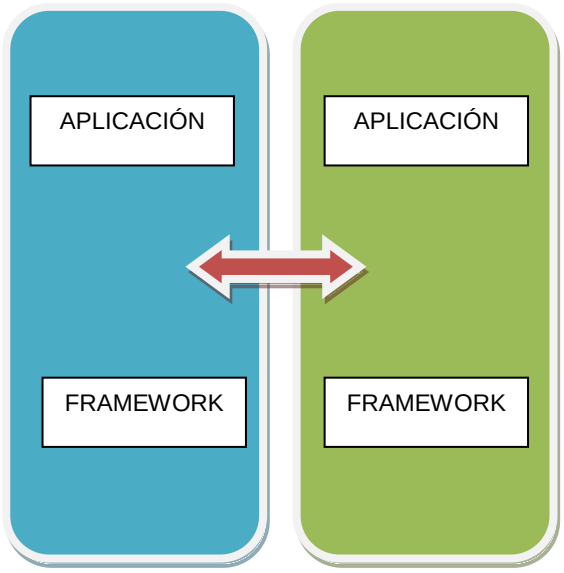
1.1) Server-Centric versus Client-Centric:

En el campo de las aplicaciones AJAX, existen dos tipos de arquitecturas de visualización, conocidas como **server-centric** (centrado en el servidor) y **client-centric** (centrado en el cliente), que determinan en qué lugar corre la aplicación web. En un framework **server-centric**, como Struts y JSF, la aplicación está alojada completamente en el servidor de aplicaciones que es el encargado de realizar todo el procesamiento de la vista. El navegador es utilizado únicamente para presentación de datos. Por el contrario, en un framework **client-centric** como GWT, las aplicaciones realizan todo su procesamiento visual en el cliente, corriendo JavaScript en el navegador y utilizando al servidor como proveedor de servicios de datos.

Server-Centric



Client-Centric



1.2) Ventajas de client-centric:

- Libera recursos del servidor.
- Escalabilidad: El uso de recursos en el servidor prácticamente no varía a pesar de que aumente el número de clientes conectados.
- Tolerancia a fallos: La aplicación no requiere reiniciarse ante caídas del servidor.
- Tiempo de respuesta reducido.
- Se requiere poco ancho de banda para interactuar con el servidor.
- Posibilidad de aplicar caché agresivo en los archivos de la vista.

1.3) Ventajas de server-centric:

- El enlazado de objetos de negocio con la vista es sencillo.
- La red es “transparente”
- El filtrado de información privada es directo.
- No exige JavaScript activado en el navegador.
- Apto para SEO.

1.4) Formatos de intercambio AJAX:

Para intercambio de datos, en las aplicaciones AJAX client-centric se suelen utilizar los formatos XML y JSON. GWT soporta ambos formatos además de un formato propio conocido como GWT RPC.

Gracias a la arquitectura client-centric, GWT no exige que el servidor soporte Java. Dado que las aplicaciones generadas corren sobre JavaScript en el cliente, GWT ni siquiera impone que el servidor soporte algún tipo de lenguaje o entorno de ejecución en particular. Esta característica permite interactuar con servidores con diferentes runtimes como PHP, Phyton, Ruby, etc., y es en estos casos en los que se utilizan los formatos XML y JSON. JSON es también utilizado cuando se requiere interactuar con servicios de internet como los que proveen Google y Yahoo, que exponen APIs de servicio bajo este formato.

Sin embargo, cuando se utilizan contenedores de Servlets, se utiliza el formato GWT-RPC, que permite intercambiar objetos Java puros por medio de un request HTTP. Esto hace que la codificación sea mucho más natural y análoga a lo que podría ser un cliente RMI.

En esta unidad exploraremos esta última posibilidad de comunicación, tomando como referencia la aplicación HolaMundo creada en la unidad anterior.

2) GWT – RPC:

El mecanismo de comunicación RPC (Remote Procedure Call), permite que el cliente invoque un método del servidor intercambiando objetos Java sobre HTTP de manera transparente, sin ningún tipo de manipulación intermedia como podría ocurrir con un servicio JSON o XML.

2.1) Deferred Binding:

Para entender el mecanismo de RPC, primero presentaremos un concepto central del compilador de GWT, conocido como Deferred Binding (Enlace Diferido), que se encarga de generar código en tiempo de compilación. Las diferentes librerías utilizan esta característica de generación de código para implementar funcionalidades que serían muy complejas de codificar manualmente. El mecanismo de RPC utiliza Deferred Binding para generar un código capaz de serializar las clases compartidas entre cliente y servidor, como veremos más adelante. Para invocar el mecanismo de Deferred Binding, se utiliza el método `GWT.create(Class)` que recibe como parámetro un literal de clase, y devuelve una instancia de una clase generada automáticamente. Utilizaremos este método con mucha frecuencia, según vayamos conociendo más funcionalidades de GWT. A medida que avancemos en el curso, nos iremos interiorizando y entendiendo mejor el concepto de Deferred Binding.

2.2) Anatomía de Servicios RPC:

Cada servicio tiene una pequeña familia de interfaces y clases asociadas. Algunas de estas clases son generadas automáticamente por medio del Deferred Binding, y generalmente nunca nos enteraremos de su existencia. El patrón de esta familia de clases es idéntico para cada servicio que implementemos.

Por cada servicio debemos definir una interface conteniendo los métodos que servirán de puntos de acceso en la comunicación del RPC. Estos métodos también se conocen como “procedimientos remotos”. Las interfaces de servicio, deben ser legibles por el compilador de GWT, por lo que deben colocarse dentro de los paquetes cliente. La interface de servicio será implementada por un Servlet especial definido por nosotros, que contendrá la implementación de cada uno de los procedimientos remotos. Esta implementación no será legible por el compilador de GWT.

Para analizar un ejemplo concreto de servicio remoto, retomaremos la aplicación HolaMundo que creamos en la unidad anterior, y revisaremos su estructura de clases.

Si exploramos el paquete client de nuestra aplicación, veremos que el asistente de proyectos ha creado una interface `GreetingService` (GreetingService.java). Esta es una interface de servicio de muestra, que contiene un único método `greetServer()`

```
public interface GreetingService extends RemoteService {
      String greetServer(String name) throws IllegalArgumentException;
}
```

El método `greetServer()` es un procedimiento remoto, que recibe como parámetro un String y devuelve otro String. Debemos notar, que esta interface extiende RemoteService, que es una interface de marcado (no contiene métodos).

2.3) RemoteServiceServlet

Por convención, la implementación de la interface de servicio `GreetingService`, se encuentra dentro del paquete **server** bajo el nombre de `GreetingServiceImpl` (GreetingServiceImpl.java).

`GreetingServiceImpl` implementa `GreetingService` y extiende a la clase `RemoteServiceServlet`, que es un tipo de servlet especial, capaz de responder a las invocaciones de RPC.

```
public class GreetingServiceImpl extends RemoteServiceServlet implements
            GreetingService {

      public String greetServer(String input) throws IllegalArgumentException {
      ...
      }
}
```

Si bien `greetServer()` corre en el servidor, es necesario invocarlo desde el cliente de alguna manera. A diferencia de Java, JavaScript posee un único hilo de ejecución, por lo que no permite ejecutar tareas concurrentes. Esto repercute en que las tareas de entrada salida, como por ejemplo la comunicación RPC, no deben bloquear el hilo actual de ejecución, porque si así lo hacen, bloquean la aplicación por completo. Esto significa que las invocaciones al servidor deben realizarse de manera asincrónica. Es por ello que el cliente de RPC debe implementar una interface complementaria de la interface de servicio, llamada interface de servicio asíncrona. En nuestro ejemplo, esta interface reside en el paquete client, bajo el nombre de `GreetingServiceAsync`.

```
public interface GreetingServiceAsync {
      void greetServer(String input, AsyncCallback<String> callback)
                  throws IllegalArgumentException;
}
```

Debemos notar que esta interface posee una versión asíncrona del método `greetServer()`. En las interfaces asíncronas, todos los métodos tienen retorno void y reciben los mismos parámetros y en el mismo orden que sus contrapartes en la interface de servicio, con el agregado de un parámetro final de tipo `AsyncCallback<T>`. `AsyncCallback` es una interface genérica con dos métodos, `onSuccess()` y `onFailure()`.

```
public interface AsyncCallback<T> {

    void onFailure(Throwable caught);

    void onSuccess(T result);
}
```

El método `onSuccess()` es invocado cuando se recibe la respuesta de RPC desde el servidor y recibe como parámetro el resultado del procedimiento remoto. El método `onFailure()` se invoca en el caso de existir errores en la comunicación, o bien, si el procedimiento remoto disparó una excepción, ésta es pasada como parámetro.

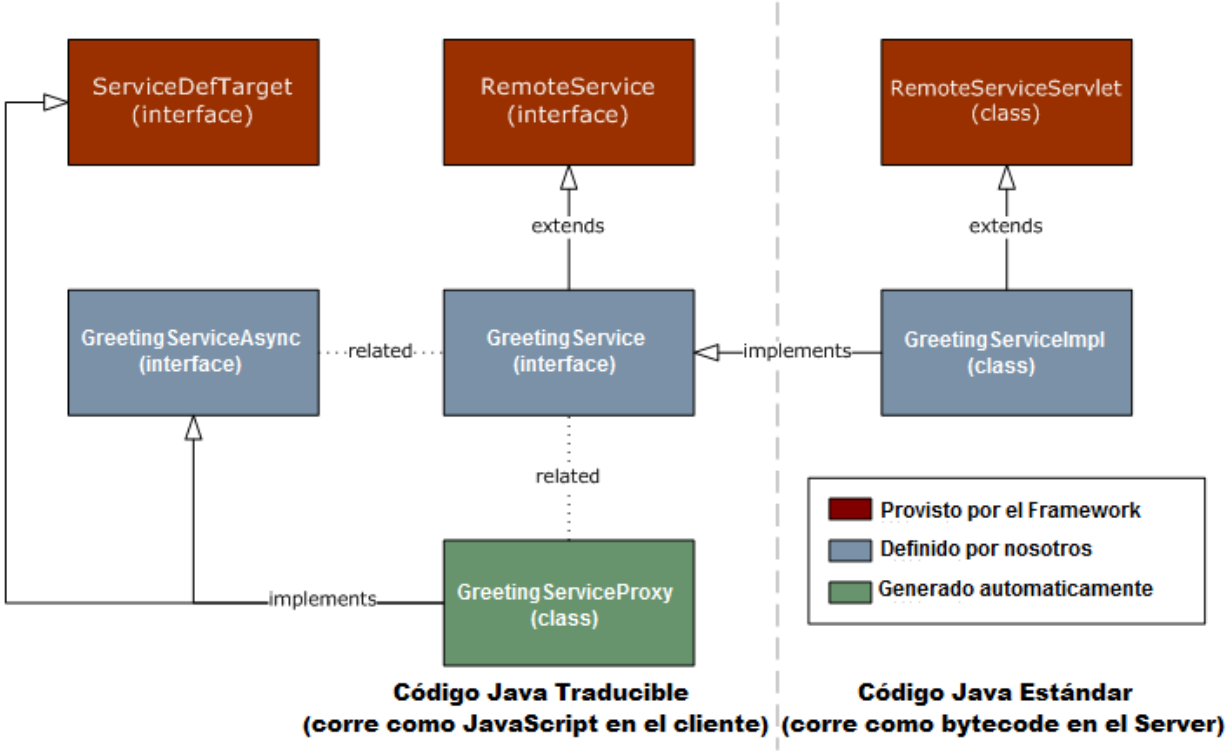
Ahora bien, tenemos un conjunto de interfaces en el cliente, pero aún ninguna implementación de las mismas. Es aquí en donde entra en juego el Deferred Binding que anteriormente presentamos, que se encarga de crear automáticamente una implementación de la interface `GreetingServiceAsync`. En la línea 35 de HolaMundo.java, vemos cómo invocar al Deferred Binding para que se encargue de crear esta implementación.

```
      private final GreetingServiceAsync greetingService = GWT
                        .create(GreetingService.class);
```

El método `GWT.create()` recibe como parámetro el literal de clase correspondiente a la interface de servicio `GreetingService`, y devuelve como resultado la implementación de `GreetingServiceAsync`. El servicio remoto es invocado en la línea 122 de HolaMundo.java, enviando como parámetro el String textToServer. En este mismo lugar podemos apreciar que `greetServer()` recibe como parámetro una implementación de `AsyncCallback<String>` como clase anónima. En la línea 124 se muestra el código para manejar los errores de comunicación en el método `onFailure()`, mientras que en la línea 135 vemos qué ocurre si el método retorna con éxito.

`GWT.create()` ha creado, por medio del Deferred Binding, una clase invisible para nosotros de nombre `GreetingServiceProxy`. Como desarrolladores, jamás podremos visualizar esa clase, sin embargo existirá a nivel de implementación.

La interfaz asíncrona debe definirse siempre en el mismo paquete que la interface de servicio, y tener el mismo nombre seguido de la palabra “Async”. Es muy importante respetar estas convenciones ya que el generador de código invocado por el Deferred Binding, se basa en ellas para generar la clase Proxy.



Ahora debería ser clara la razón por la cual debemos definir `GreetingService` dentro de un paquete visible al compilador, ya que su literal de clase se requiere como parámetro de `GWT.create()`.

Como cada implementación de servicio se define como un Servlet, es necesario configurarla en el archivo web.xml, y proveerle una URL especial. Si revisamos la línea 9 de web.xml, veremos que `GreetingServiceImpl` se define como servlet bajo el nombre greetServlet.

```
<servlet>
  <servlet-name>greetServlet</servlet-name>
  <servlet-class>
      com.logikas.eduka.handsongwt.holamundo.server.GreetingServiceImpl
  </servlet-class>
</servlet>
```

En la línea 14 de web.xml se define el mapeo de greetServlet hacia la URL /holamundo/greet.

```
<servlet-mapping>
  <servlet-name>greetServlet</servlet-name>
  <url-pattern>/holamundo/greet</url-pattern>
</servlet-mapping>
```

El navegador invoca al RPC por medio del método POST de HTTP sobre la URL configurada en el archivo web.xml (/holamundo/greet en nuestro caso). Por este motivo, el navegador necesita conocer explícitamente sobre qué URL está configurado el servlet, ya que no posee acceso al archivo web.xml. Esta configuración se realiza por medio de la anotación `RemoteServiceRelativePath`, como podemos ver en la línea 9 de GreetingService.java.

```
@RemoteServiceRelativePath("greet")
```

2.4) Reutilización de código en el cliente:

En el paquete shared se encuentra la clase `FieldVerifier` (FieldVerifier.java), que posee un único método `isValidName()`, cuya finalidad es verificar el valor del campo de texto ingresado. Esta clase, es utilizada tanto por la clase `GreetingServiceImpl` como por la clase `HolaMundo`.

GreetingServiceImpl.java – línea 16

```
if (!FieldVerifier.isValidName(input)) {
    // If the input is not valid, throw an IllegalArgumentException back to
    // the client.
    throw new IllegalArgumentException(
        "Name must be at least 4 characters long");
}
```

HolaMundo.java – línea 113

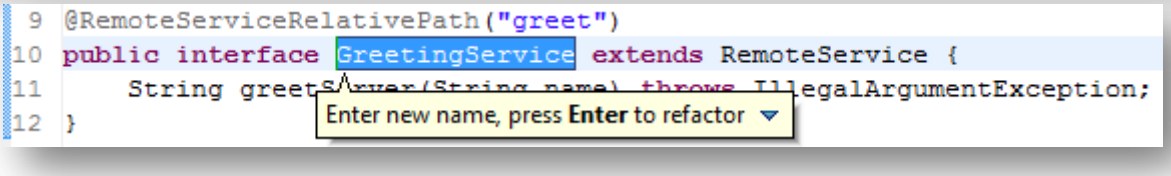
```
if (!FieldVerifier.isValidName(textToServer)) {
    errorLabel.setText("Please enter at least four characters");
    return;
}
```

Es por eso que la clase `FieldVerifier` se encuentra en el paquete shared (compartido), ya que es utilizada tanto en cliente como en servidor. Esto demuestra una de las posibilidades más potentes de GWT, que es la reutilización de código, sin necesidad de reescribir las funcionalidades de Java funciones en JavaScript.

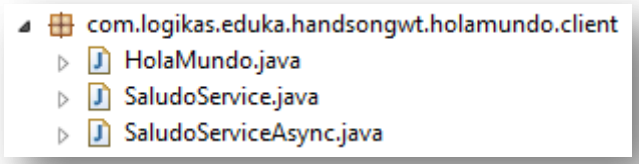
2.5) Refactorización de la interface de servicio.

Como vimos anteriormente, la interface asíncrona debe mantener su nombre y sus métodos de servicio en correlación con la interface de servicio. Esto supone una carga extra al trabajo de mantenimiento, ya que si cualquier cambio en la interface de servicio debe ser reflejado en la interface asíncrona. Afortunadamente, el Google Eclipse Plugin, nos ayuda a mantener la sincronización del código.

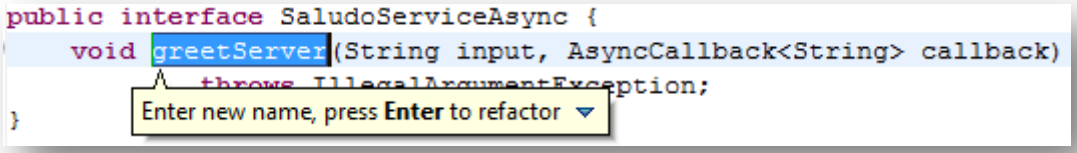
Para comprobar esta característica, posicionaremos el cursor sobre el nombre de la interface `GreetingService` en la línea 10 de GreetingService.java, y presionaremos Alt+Shit+R para renombrar la clase por medio del refactorizador de Eclipse.



Ahora colocaremos el nombre `SaludoService`, y presionaremos Enter. Veremos que esta acción, cambió el nombre de `GreetingServiceAsync` por `SaludoServiceAsync`.



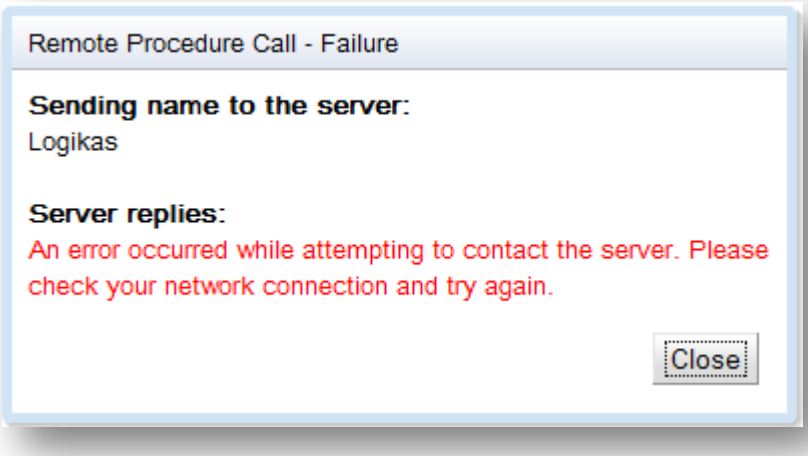
Haremos algo similar con el método `greetServer()`, al cual le cambiaremos el nombre por `saludar()`, pero para comprobar que la refactorización de servicios es simétrica, cambiaremos el nombre de este método en la clase `SaludoServiceAsync`. Esto hará que el método relacionado en `SaludoService` cambie también de nombre.



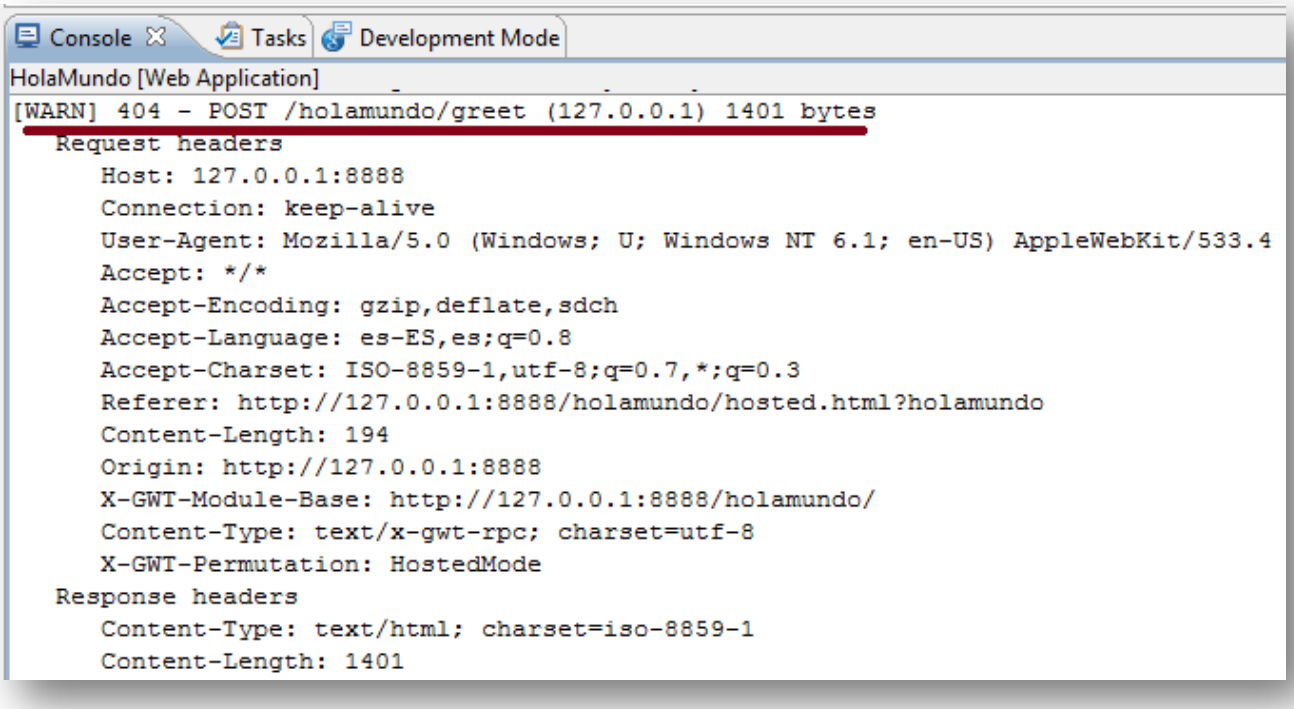
A pesar de la refactorización que hicimos, podemos ver que el nombre de la clase `GreetingServiceImpl` no fue alterado. Esto se debe a que no es necesario que el nombre de la clase que implementa el servicio esté correlacionado con el de la interface de servicio. Nosotros cambiaremos su nombre por `SaludoServiceImpl` para mantener la legibilidad. Recordemos que al renombrar este servlet, debemos también cambiar el nombre de la clase en el descriptor de servlet que se encuentra en la línea 11 del archivo web.xml. También cambiaremos el nombre del servlet por "*saludoServlet*", por lo que deberemos actualizarlo en la etiqueta `servlet-mapping`. También cambiaremos la URL del servlet, cambiando "*greet*" por "*saludo*".

```
<servlet>
  <servlet-name>saludoServlet</servlet-name>
  <servlet-class>
    com.logikas.eduka.handsongwt.holamundo.server.SaludoServiceImpl
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>saludoServlet</servlet-name>
  <url-pattern>/holamundo/saludo</url-pattern>
</servlet-mapping>
```

Ahora correremos nuestra aplicación HolaMundo en Development Mode, colocaremos nuestro nombre sobre la caja de texto, y haremos click en “Enviar”. Para nuestra sorpresa, la aplicación nos mostrará que hubo un error de comunicación.



Para entender de qué se trata este error, observaremos la consola de mensajes de Eclipse. La consola nos muestra el encabezado HTTP generado a raíz de la petición realizada. En la primera línea vemos que se trata del mensaje de error HTTP 404 (no encontrado), generado a partir de la petición de POST a la URL /holamundo/greet. Esto significa que, a pesar de que cambiamos la URL /holamundo/greet por /holamundo/saludo en el `servlet-mapping` del archivo web.xml, el cliente sigue pidiendo por la URL antigüa.



Para solucionar este problema, debemos editar la línea 9 de SaludoService.java, y cambiar el string "*greet*" por "*saludo*" en la anotación `RemoteServiceRelativePath`.

```
@RemoteServiceRelativePath("saludo")
```

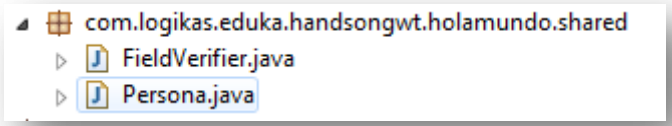
Guardaremos el cambio, refrescaremos la ventana del navegador e intentaremos nuevamente enviar nuestro nombre. Si hemos seguido correctamente cada uno de los pasos anteriores, la aplicación debería devolvernos el mensaje de saludo habitual sin inconvenientes.

2.6) Serialización de objetos a través del RPC:

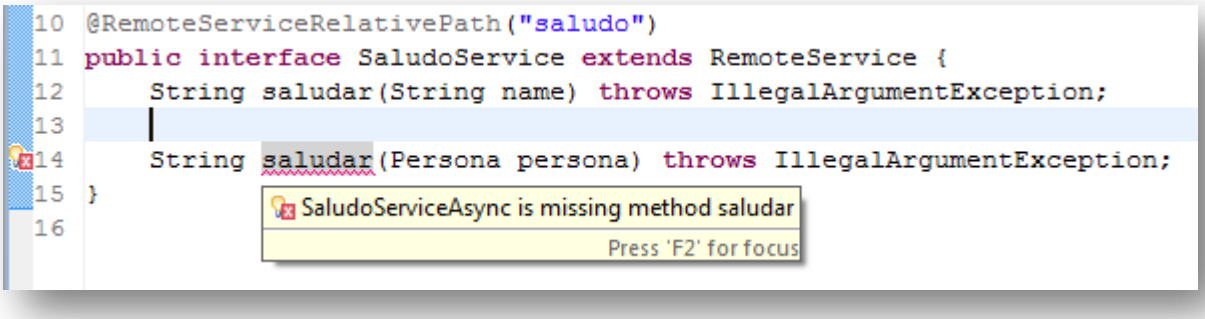
Hasta ahora hemos intercambiado sólo cadenas de texto en nuestra comunicación con el servidor. Sería interesante explorar la posibilidad de enviar algo más complejo que una cadena, como por ejemplo, un objeto que represente a una persona, en lugar de tan sólo un nombre. Para ello crearemos una clase Persona que contenga una propiedad nombre de un constructor por defecto y un constructor que reciba como parámetro el nombre.

```
public class Persona {

  private String nombre;

  public Persona(String nombre) {
    this.nombre = nombre;
  }


  public Persona() {
    this("");
  }

  public String getNombre() {
    return nombre;
  }

  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
}
```

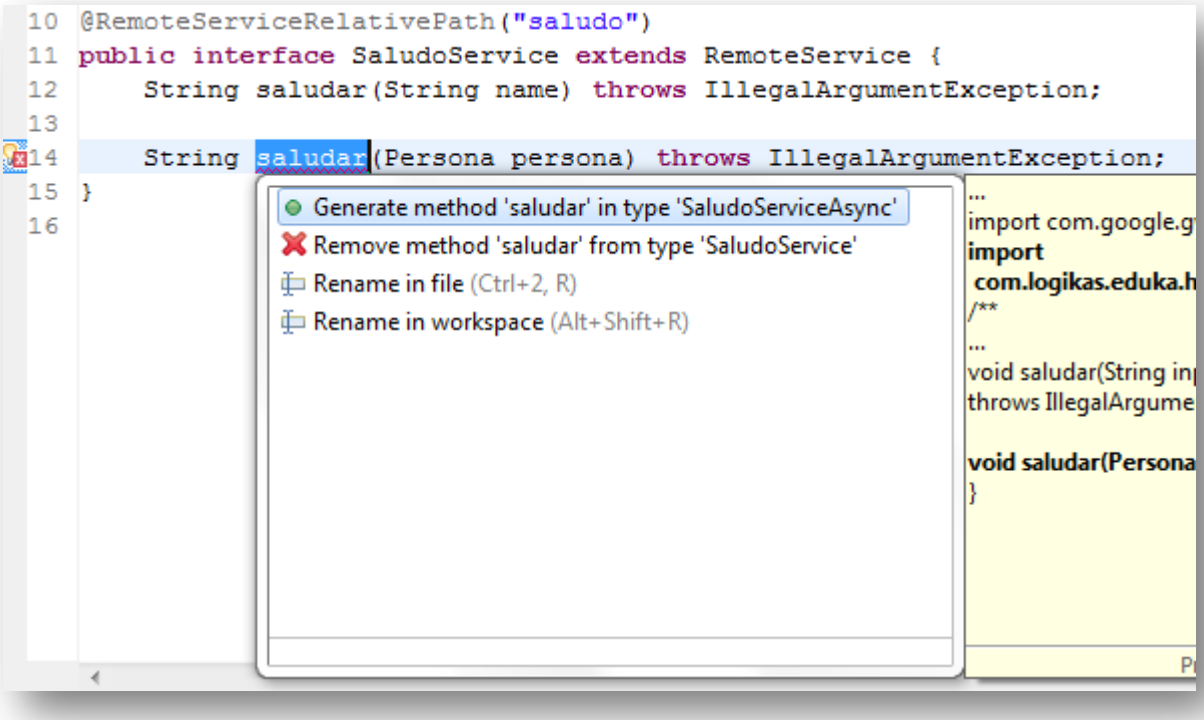
Esta clase deberá situarse en el paquete shared, ya que será compartida tanto por cliente como por servidor.



Luego agregaremos en la interface de servicio (SaludoService.java) un nuevo método `saludar()`, que recibirá como parámetro un objeto persona. Para nuestra sorpresa, Eclipse está marcando un error en el método agregado. Gracias al Google Plugin, Eclipse detecta que no existe el método correspondiente en la interface asíncrona.



Para remediar este problema, dejaremos que Eclipse lo resuelva aceptando la sugerencia de generar el método `saludar()` en la interface asíncrona.



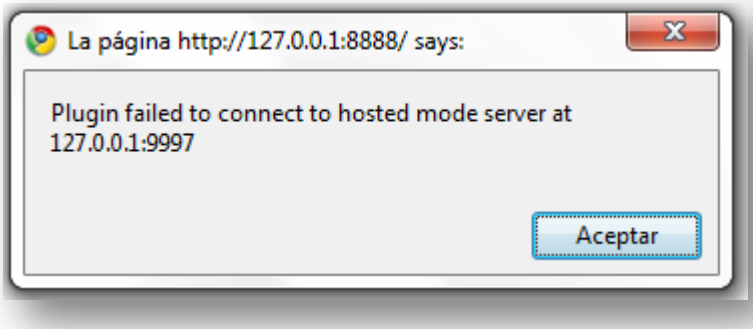
Luego escribiremos la implementación del nuevo método `saludar()` en la implementación del servicio (SaludoServiceImpl.java).

```
@Override
public String saludar(Persona persona) throws IllegalArgumentException {
      return saludar(persona.getNombre());
}
```

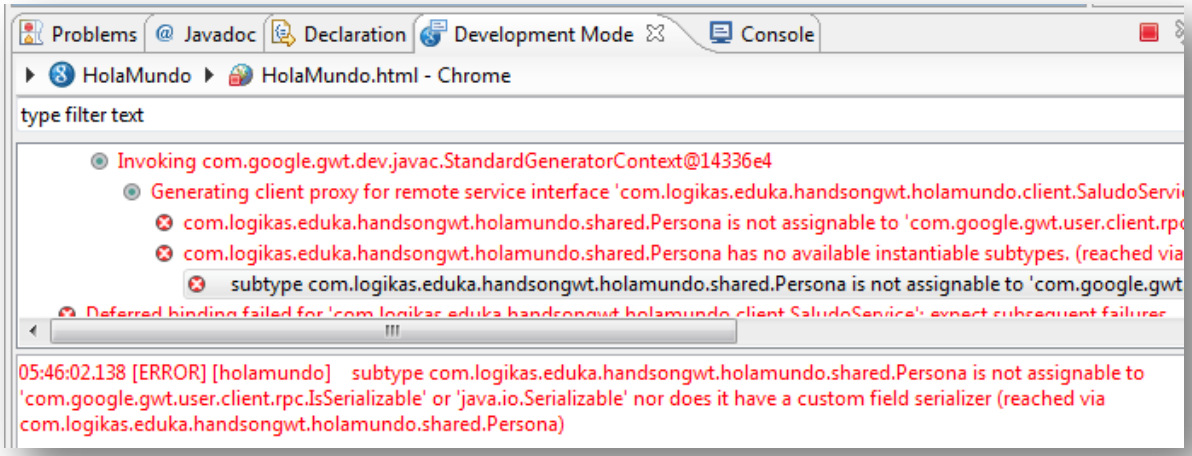
Por último, cambiaremos la línea 123 de HolaMundo.java, instanciando un objeto Persona en la invocación del método saludar().

```
greetingService.saludar(new Persona(textToServer),
```

Luego intentaremos probar este cambio, refrescando la ventana del navegador. Para nuestra sorpresa, el navegador nos muestra un mensaje de error, que nos dice que es imposible comunicarse con el servidor.



Si revisamos la consola del Development Mode, observaremos que nos muestra un árbol de errores. Si hacemos click sobre cada ítem de este árbol, podremos leer su detalle en el panel inferior.



Nos interesa el error más anidado, que nos indica que la clase `Persona` no es asignable a `com.google.gwt.user.client.rpc.IsSerializable`.

2.6.1) Tipos Serializables.

La serialización es un proceso que permite codificar un objeto residente en memoria, en un formato capaz de ser transportado a través de una red de datos. Este proceso es bien conocido en el universo de Java por su utilización en la comunicación RMI. En el caso del RPC de GWT, los objetos son serializados hacia una cadena de texto, para poder ser transportados a través de HTTP hacia el servidor y viceversa. Debido a que sólo un conjunto reducido de las clases de nuestra aplicación serán transportadas a través de la red, es necesario individualizarlas de alguna manera, para que GWT las reconozca como clases serializables. Es por esto que existe la interface de marcado IsSerialiable, que debe ser implementada por cualquier clase que requiera ser transportada a través del RPC.

Como nuestra clase Persona no implementa la interface `com.google.gwt.user.client.rpc.IsSerializable`, GWT no la reconoce como clase transportable, y por ello falla la carga de nuestra aplicación.

Para remediar este problema, haremos que Persona implemente IsSerializable, y entonces podremos correr nuestra aplicación sin inconvenientes.

```
import com.google.gwt.user.client.rpc.IsSerializable;

public class Persona implements IsSerializable
```

2.6.2) Requisitos de los objetos serializables.

- Todos los objetos serializables requieren un constructor por defecto, con visibilidad pública, protegida o por default.
- Los campos que no deban transportarse, deben ser marcados con el modificador **transient**.
- Los campos no transient deben ser de tipos primitivos o serializables.
- Los campos no transient no pueden estar marcados por el modificador **final**.
- La clase java.lang.Object no es serializable.

2.6.3) Interface java.io.Serializable.

Por compatibilidad con Java, el RPC de GWT también permite utilizar como alternativa a IsSerializable, la interface Serializable, que es de uso estandarizado en Java. Sin embargo, en la medida de lo posible, es preferible utilizar IsSerializable, ya que GWT presenta algunas restricciones hacia la interface Serializable.

El mecanismo de comunicación RPC es el más utilizado en una aplicación GWT, por eso es importante comprenderlo en su totalidad. A medida que avancemos con el curso, conoceremos patrones de uso de este mecanismo en relación con la interface de usuario.

Conclusión:

En esta unidad hemos comprendido las arquitecturas client-centric y server-centric, que nos fue útil para presentar las posibilidades de comunicación con el servidor que ofrece GWT. Desde los formatos estándar como JSON y XML, hasta el sofisticado GWT RPC. Adicionalmente, hemos tomado un primer contacto con el Deferred Binding, que es una característica central del compilador GWT que será de mención frecuente en unidades futuras.

Enlaces de Interés:

- Guia oficial de comunicación GWT con el servidor: http://code.google.com/webtoolkit/doc/latest/DevGuideServerCommunication.html
- Historia de la cerveza, la esposa y el programa de TV: http://groups.google.com/group/google-web-toolkit/browse_thread/thread/faca1575f306ba0f/bd72c28afe5eb680